Lecture 10: Key Distribution for Symmetric Key Cryptography and Generating Random Numbers

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 10, 2017
9:33am

Goals:

- Why might we need **key distribution centers**?
- **Master key** vs. **Session key**
- The **Needham-Schroeder** and **Kerberos** Protocols
- Generating **pseudorandom** numbers
- Generating **cryptograhically secure pseudorandom** numbers
- Hardware and software entropy sources for **truly random** numbers
- **A word of caution regarding software entropy sources**

CONTENTS

10.1: THE NEED FOR KEY DISTRIBUTION CENTERS

- Let's say we have a large number of people, processes, or systems that want to communicate with one another in a secure fashion. Let's further add that this group of people/processes/systems is not static, meaning that the individual entities may join or leave the group at any time.

- A simple-minded solution to this problem would consist of each party physically exchanging an encryption key with every one of the other parties. Subsequently, any two parties would be able to establish a secure communication link using the encryption key they possess for each other. **This approach is obviously not feasible for large groups of people/processes/systems, especially when group membership is ever changing.**

- A more efficient alternative consists of providing every group member with a single key for securely communicate with a **key distribution center** (KDC). This key would be called a **master key**. When A wants to establish a secure communication link with B, A requests a **session key** from KDC for communi-

cating with B.

- In implementation, this approach must address the following issues:
  - Assuming that A is the initiator of a session-key request to KDC, when A receives a response from KDC, how can A be sure that the sending party for the response is indeed the KDC?
  - Assuming that A is the initiator of a communication link with B, how does B know that some other party is not masquerading as A?
  - How does A know that the response received from B is indeed from B and not from someone else masquerading as B?
  - What should be the lifetime of the session key acquired by A for communicating with B?
- The next section presents how the Needham-Schroeder protocol addresses the issues listed above. A more elaborate version of this protocol, known as the Kerberos protocol, will be presented in Section 10.3.

10.2: THE NEEDHAM-SCHROEDER KEY DISTRIBUTION PROTOCOL

A party named A wants to establish a secure communication link with another party B. Both the parties A and B possess **master keys** K_A and K_B, respectively, for communicating privately with a **key distribution center** (KDC). [In a university setting, there is almost never a need for user-to-user secure communication links. So for folks like us in a university, all we need is a password to log into the computers. However, consider an organization like the U. S. State Department where people working in different U.S. embassies abroad may have a need for user-to-user secure communication links. Now, in addition to the master key, a user named A may request a session key for establishing a direct communication link with another user named B. This session key, specific to one particular communication link, would be valid only for a limited time duration. This is where Needham-Schroeder protocol can be useful.] Now A engages in the following protocol (Figure 1):

- Using the key K_A for encryption, user A sends a request to KDC for a **session key** intended specifically for communicating with user B.

- The message sent by A to KDC includes A's network address (ID_A), B's network address (ID_B), and **a unique session identifier**. The session identifier is a **nonce** — short for a "number used once" — and we will denoted it N_1. The primary requirement on a nonce — a random number — is that it be unique to each request sent by A to KDC. The message sent by A to KDC can be expressed in shorthand by

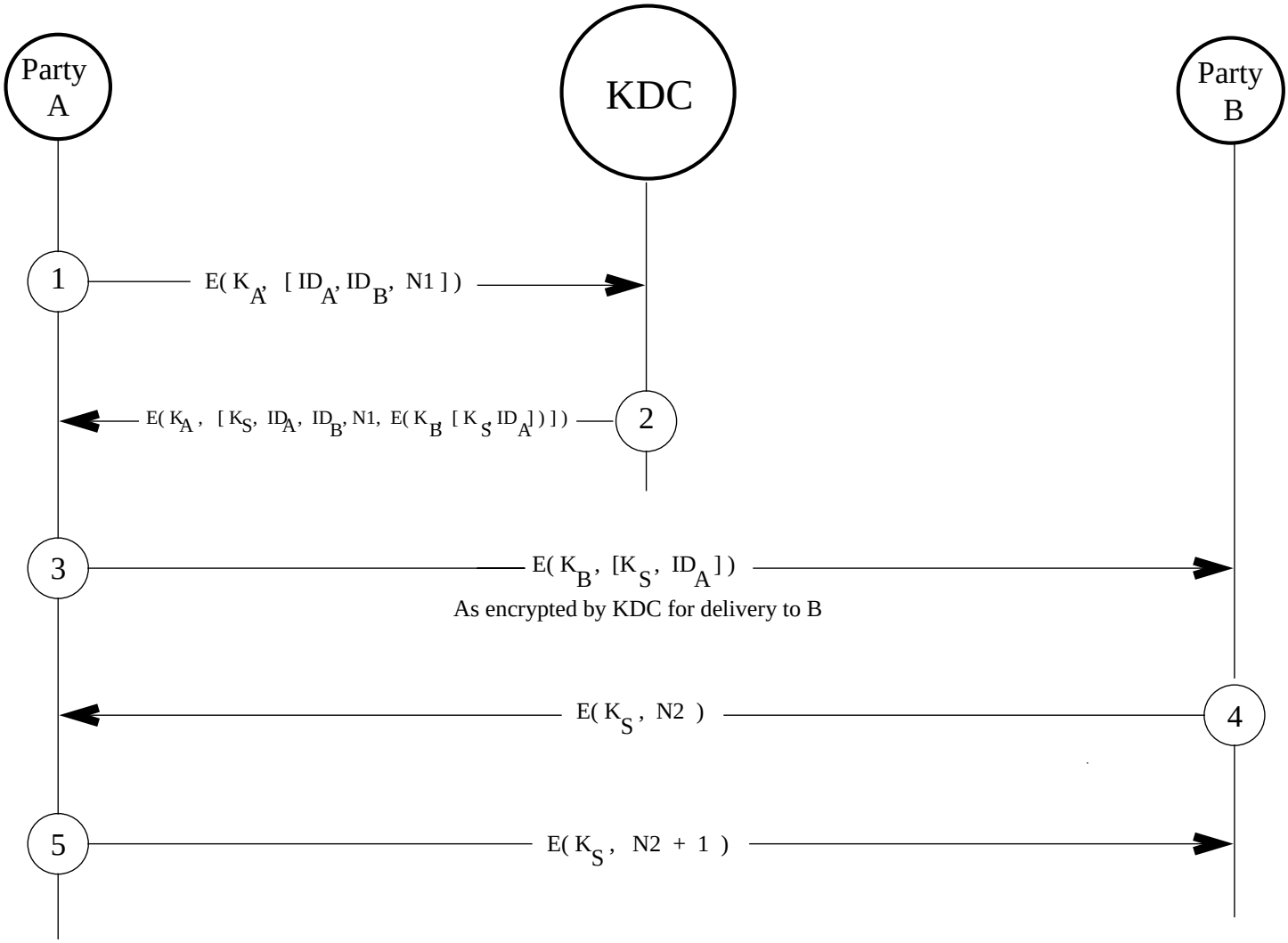

Figure 1: *A pictorial depiction of the Needham-Schroder protocol.* (*This figure is from Lecture 10 of "Computer and Network Security" by Avi Kak*)

$$E(K_A, \quad [ID_A, \ ID_B \ , N1])$$

where $E(.,.)$ stands for encryption of the second-argument data block with a key that is in the first argument.

- KDC responds to A with a message encrypted using the key K_A. The various components of this message are

  – The session-key K_S that A can use for communicating with B.

  – The original message received from A, including the nonce used by A. This is to allow A to match the response received from KDC with the request sent. Note that A may be trying to establish multiple simultaneous sessions with B.

  – A "packet" of information meant for A to be sent to B. This packet of information, encrypted using B's master key K_B includes, again, the session key K_S, and A's identifier ID_A. (Note that A cannot look inside this packet because A does **not** have access to B's master key K_B. We will sometimes refer to this packet of information as a **ticket** that A receives for sending to B.

- The message that KDC sends back to A can be expressed as

$$E(K_A, \quad [K_S, \ ID_A, \ ID_B \ , N1, \ E(K_B, \ [K_S, \ ID_A]) \ ])$$

- Using the master key K_A, A decrypts the message received from KDC. Because only A and KDC have access to the master key K_A, A is certain that the message received is indeed from KDC.

- A keeps the session key K_S and sends the packet intended for B to B. This message is sent to B unencrypted by A. But note that it was previously encrypted by KDC using B's master key K_B. **Therefore, this first contact from A to B is protected from eavesdropping.**

- B decrypts the message received from A using the master key K_B. B compares the ID_A in the decrypted message with the sender identifier associated with the message received from A. By matching the two, B makes certain that no one is masquerading as A.

- B now has the session key for communicating securely with A.

- Using the session key K_S, B sends back to A a nonce $N2$. A responds back with $N2 + 1$, using, of course, the same session key K_S. This serves as a confirmation that the session key K_S works for the ongoing session — this requires that what A encrypts with K_S be different from what B encrypted with K_S. This part of the handshake also ensures that B knows that it did not receive a first contact from A that A is no longer interested in. An additional benefit of this step is that it provides some protection against a replay attack. [A replay attack takes different forms in different contexts. For example, in the situation here, if A was allowed to send back to B the same nonce that it received from the latter, then B could suspect that some other party C posing as A was merely "replaying" back B's message that it had obtained by, say, eavesdropping. In another version of the replay attack, an attacker may repeatedly send an information packet to a victim hoping to elicit from the latter variations on the response that the attacker may then analyze for some vulnerability in the victim's

machine. The PTW attack on WEP that you saw in Section 9.8.3 of Lecture 9 is an example of that form of a replay attack.] The message sent by B back to A can be expressed as

$$E(K_S,\ [N2])$$

And A's response back to B as

$$E(K_S,\ [N2\ +\ 1])$$

- This exchange of message is shown graphically in Figure 1. **A most important element of this exchange is that what the KDC sends back to A for B can only be understood by B.**

10.2.1: Some Variations on the KDC Approach to Key Distribution

- It is not practical to have a single KDC service very large networks or network of networks.

- One can think of KDC's organized hierarchically, with each local network serviced by its own KDC, and a group of networks serviced by a more global KDC, and so on.

- A local KDC would distribute the session keys for secure communications between users/processes/systems in the local network. But when a user/process/system desires a secure communication link with another user/process/system in another network, the local KDC would communicate with a higher level KDC and request a session key for the desired communication link.

- Such a hierarchy of KDCs simplifies the distribution of master keys. A KDC hierarchy also limits the damage caused by a faulty or subverted KDC.

- Before ending this section, it is important to point out that for small networks there does exist an alternative to the KDC based

approach to session-key generation. The alternative consists of storing at every node of a network the "master" keys needed for communicating privately with each of the other N nodes in a network. Therefore, each node will store $N-1$ such keys. If the messages shuttling back and forth in the network are short, you may use these keys directly for encryption. However, when the messages are of arbitrary length, a node A in the network can use the master key for another node B to first set up a session key and subsequently use the session key for the actual encryption of the messages.

10.3: KERBEROS

- To see a need for this protocol, consider the following application "scenario:"

  – Let's say that a university computer network wants to provide printer services to its students. The printers are located at certain designated locations on the campus. Each student gets a "printer budget" on a semester basis. A student is allowed a certain number of free pages. When a student has used up his/her printer budget, he/she is expected to deposit money in the registrar's office for additional pages.

  – The printers are connected to machines that we can refer to as "printer servers" that — let's say — run the CUPS software. [In Linux/Unix environments, CUPS is probably the most popular software package used today to turn your machine into printer server. (The acronym started out as standing for "Common Unix Printing System" but now it's a name unto itself. With CUPS installed, your machine can accept print requests from other hosts in your LAN or even in the internet at large if you enable CUPS accordingly. Most of the time, though, most folks use CUPS on a standalone basis to send jobs to printers that you are authorized to access.) CUPS is an implementation of the Internet Printing Protocol (IPP). Think of IPP in the same way as you think of HTTP: IPP is a client-server protocol in which the client hosts send requests for print jobs (*and, only the requests, since, eventually, the print jobs go directly to the printers*) to the server hosts. The clients may query a server for the status of a printer, for the status of a print job, the printer options, etc. In the same manner as HTTP, IPP is described in a series of RFC documents issued by IETF. For example, RFC 2910 and 2911 describe the version IPP/1.1 of the protocol. By the way, the default server port for IPP is TCP/631 in the same manner as the default server port for HTTP is TCP/80. CUPS also uses

the port UDP/631 for printer discovery. With regard to the relationship between IPP and TCP, IPP is in the application layer of the 4-layer TCP/IP stack you'll see in Lecture 16. Under IPP, each printer gets its own IP address and communications with the printer are based on the TCP protocol described in Lecture 16.]

– There are several security and authentication issues involved in this scenario. When a print request is received, a printer server must first authenticate the client — since not all the client hosts on the campus may be authorized to send jobs to the printer in question. Subsequently, the printer server must also validate the print request received against the print budget for the student who sent the request. Finally, the printer server must somehow enable a confidential communication link directly from the host where the print request originated to the printer in question. That is, you would not want a student to send his/her job to the printer in plaintext. At the same time, you would not want an off-the-shelf printer to have to do too much security-related computing **for an encrypted link between the student and the printer**. The printer server would not want to route all the print jobs through itself since that would unnecessarily bog down the server.

– The big issue here is how to establish a direct authenticated and confidential communication link between the host where the print request originated and the printer. Since printers generally are rudimentary when it comes to general purpose computing, you may not expect a printer to contain all of the software that generally is required these days (such as the SSL/TLS libraries) for establishing such links. And you certainly would not want the students to establish password based connections with the printers (for authentication) since such passwords are likely to be transmitted in clear text over a network.

– All of the difficulties mentioned above are solved by the Kerberos

protocol described in this section. With the Kerberos protocol, there is no reason to transmit passwords in clear text or otherwise. As in the Needham-Schroeder protocol, Kerberos operates on the principle of shared secret keys. If you have enabled Kerberos in the CUPS software on the printer server, when you add a client host to the group of hosts allowed to send print jobs to the printers, you'll simultaneously create a secret key (like the master keys in the Needham-Schroeder protocol) that will be shared by the client and the printer server. The printer server will also possess a shared secret key for communicating with each of the printers it is in charge of. Eventually, through the Kerberos protocol, the printer server will bring into existence a secret session key that would allow, say, **a student's laptop to send a print job directly to the printer over an encrypted link.**

- This protocol provides security for client-server interactions in a network. We are talking about servers such as printer servers (as in the example described above), database servers, news servers, FTP servers, and so on.

- The main difference between the Needham-Schroeder protocol and the Kerberos protocol is that the latter makes a distinction between the clients, on the one hand, and the service providers, on the other. As you will recall, no such distinction is made in the Needham-Schroeder protocol.

- In the Kerberos protocol, the Key Distribution Center (KDC) is divided into two parts, one devoted to client authenticaiton, and the other in charge of providing security to the service providers.

- With regard to the strange sounding name of this protocol, note that Kerberos is another name for Cerberus, the three-headed dog who guards the gates of Hades in Greek mythology.

- As mentioned, and as shown in Figure 2, the KDC in Kerberos has two parts to it, one in charge of security vis-a-vis the clients and the other in charge of the security vis-a-vis the service providers. The former is called the Authentication Server (AS) and the latter the Ticket Granting Server (TGS). If there is any complexity to Kerberos, especially vis-a-vis the Needham-Schroeder protocol, it is owing to the fact that a client cannot gain direct access to TGS and only the TGS can provide a session key to communicate with a service provider. A client must first authtenticate himself/herself/itself to AS and obtain from AS a session key for accessing TGS. [Consider again the printer scenario I painted for when Kerberos is supposed to be used — it makes perfect sense in that scenario to separate the Authentication Server AS from the Ticket Granting Server TGS. With such a separation, AS can concern itself exclusively with matters related to user authentication, which would include keeping up-to-date with who is allowed to use which printers. In a large enterprise like Purdue University with its tens of thousands of users, the database of users and which printers and other resources the users are allowed to access is bound to be in a constant state of flux as new students join the university, graduating students leave, and other students who may drop out for a while. On the other hand, TGS can concern itself exclusively with issues related directly to the printers. These issues could include keeping track of the current load on each printer so that a user wanting access to a printer that already has too many jobs in its queue could be warned; etc.]

- In the rest of the Kerberos protocol described here, we will use the following notation:

– K_{Client} denotes the secret key held by AS for the Client.

– K_{TGS} denotes the secret key held by AS for TGS. TGS also has this key.

– $K_{ServiceProvider}$ denotes the secret key held by AS for the Service Provider. The Service Provider also has access to this key.

– $K_{Client-TGS}$ denotes the **session key** that AS will send to the Client for communicating with TGS.

– $K_{Client-ServiceProvider}$ denotes the **session key** that TGS will send to the Client for communicating with the Service Provider.

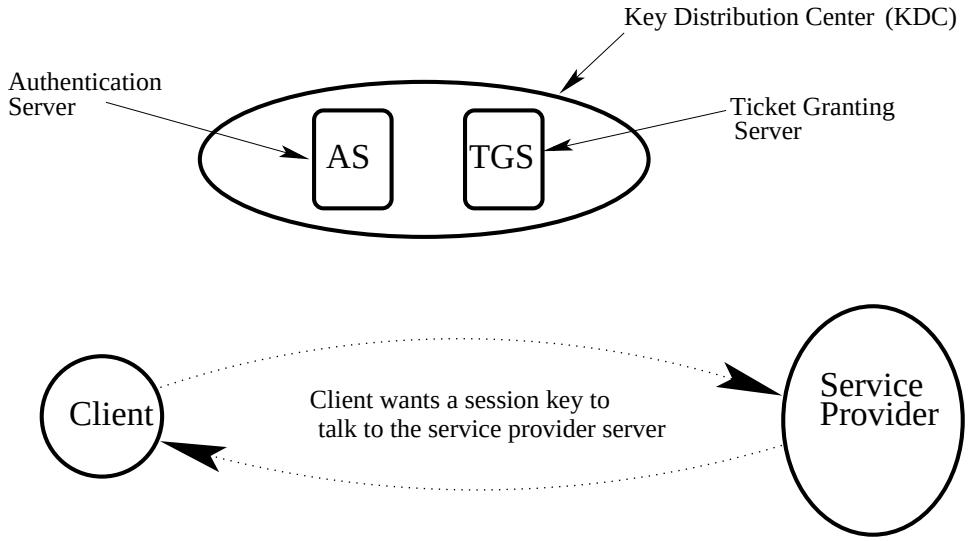

Figure 2: *The main "actors" that participate in the Kerberos protocol.* (*This figure is from Lecture 10 of "Computer and Network Security" by Avi Kak*)

- Each Client *registers* with the Authentication Server and is granted a user identity and a secret password. As shown in Figure 3, the Client sends a request *in plain text* to the AS. This request is for a service that the Client expects from the Service Provider. (Message 1) [The message numbers are shown in small circles in Figure 3.]

- The AS sends back to the Client the following two messages encrypted with the K_{Client} key. In the database maintained by AS, this key is specific to the Client and will remain unchanged as long as the client does not alter his/her password. Note that this encryption key is **not** directly known to the Client. The two messages are:

  – A session key $K_{Client-TGS}$ that the client can use to communicate with TGS. (Message 2) This message may be expressed as

  $$E(K_{Client}, [K_{Client-TGS}])$$

  – A **Ticket-Granting Ticket** (TGT) that is meant for delivery to TGS. This ticket includes the client's user ID, the client's network address, validation time, and the same $K_{Client-TGS}$ session key as mentioned above. The ticket is encrypted with the K_{TGS} secret key that the AS server maintains for TGS. (Message 3) Therefore, this message from AS to the Client may be expressed by

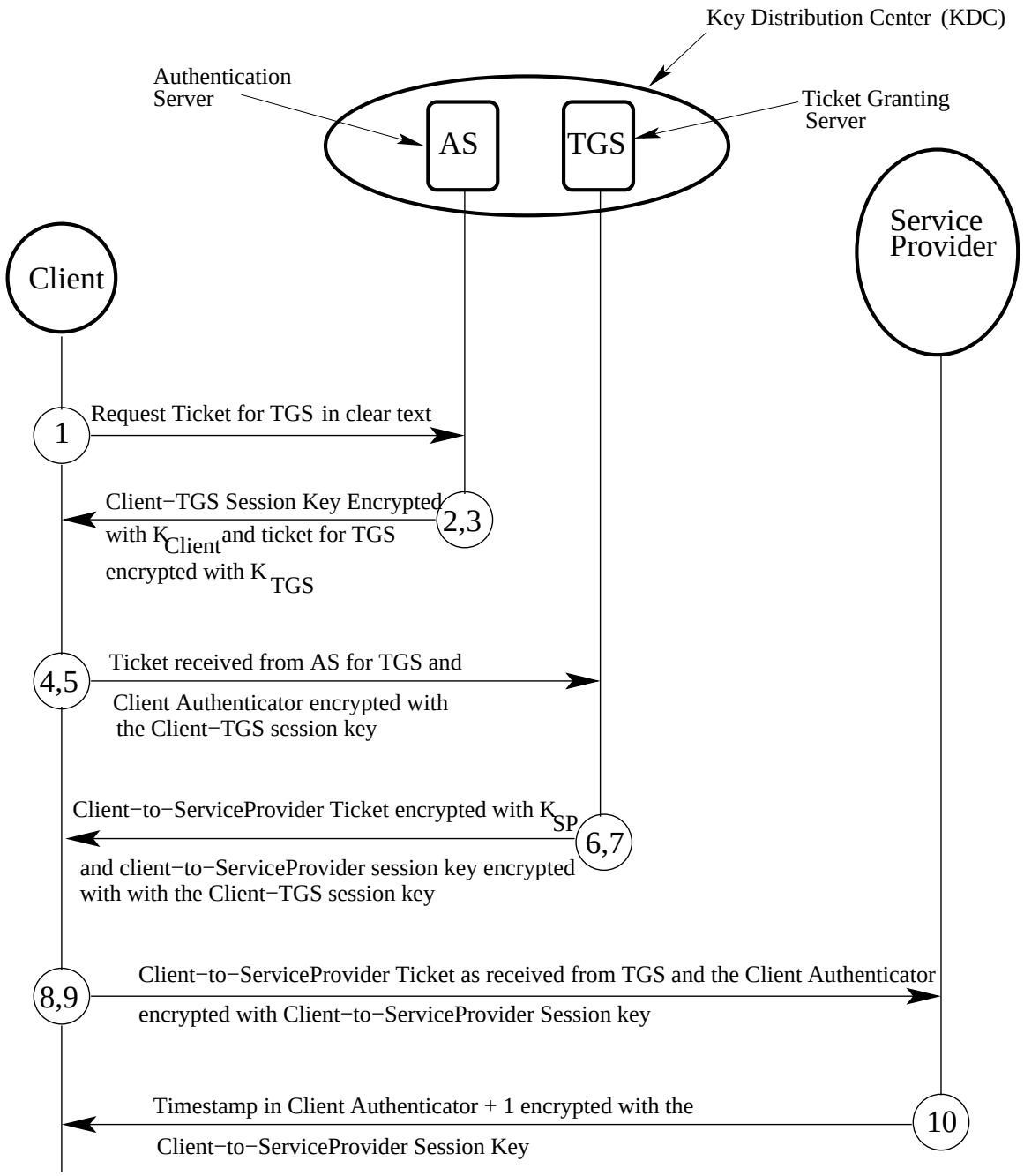

Figure 3: *A pictorial depiction of the Kerberos protocol.*

(This figure is from Lecture 10 of "Computer and Network Security" by Avi Kak.)

$E(K_{Client}, [K_{Client-TGS}, E(K_{TGS}, [ClientID, ClientIP, ValidityPeriod, K_{Client-TGS}])])$

The TGT is also referred to as the **initial ticket** since it enable the Client to subsequently obtain Client-to-ServiceProvider tickets from TGS.

- When the client receives the above messages, the client enters his/her password into a dialog box. An algorithm converts this password into what would be the K_{Client} encryption key if the password is correct. **The password is immediately destroyed** and the generated key used to decrypt the messages received from AS. The decryption allows the Client to extract the session key $K_{Client-TGS}$ and the ticket meant for TGS from the information received from AS.

- The client now sends the following two messages to TGS:

  – The encrypted ticket meant for TGS followed by the ID of the requested service. If the client wants to access an FTP server, this would be the ID of the FTP server. (Message 4)

  – A **Client Authenticator** that is composed of the client ID and the timestamp, the two encrypted with the $K_{Client-TGS}$ session key. (Message 5)

- TGS recovers the ticket from the first of the two messages listed above. From the ticket, it recovers the $K_{Client-TGS}$ session. The TGS then uses the session key to decrypt the second message listed above that allows it to authenticate the Client.

- TGS now sends back to the Client the following two messages:

  – A Client-to-ServiceProvider ticket that consists of 1) the Client ID, 2) the Client network address, 3) the validation period, and 4) a session key for the Client and the Service Provider, $K_{Client-ServiceProvider}$. This session key is encrypted with the $K_{ServiceProvider}$ key that is known to TGS. (Message 6)

  – The same $K_{Client-ServiceProvider}$ session key as mentioned above but this time encrypted with the $K_{Client-TGS}$ session key. (Message 7)

- The client recovers the ticket meant for the service provider with the $K_{Client-TGS}$ session key.

- The client next sends the following two messages to the service provider:

  – The Client-to-ServiceProvider ticket that was encrypted by TGS with the $K_{ServiceProvider}$ key. (Message 8)

  - An authenticator that consists of the Client ID and the timestamp. This authenticator is encrypted with the $K_{Client-ServiceProvider}$ session key. (Message 9)

- The Service Provider decrypts the ticket with its own $K_{ServiceProvider}$ key. It extracts the $K_{Client-ServiceProvider}$ session key from the ticket, and then uses the session key the decrypt the second message received from the client.

- If the client is authenticated, the ServiceProvider sends to the Client a message that consists of the timestamp in the authenticator received from the Client plus one. This message is encrypted using the $K_{Client-ServiceProvider}$ session key. (Message 10)

- The client decrypts the message received from the Service Provider using the $K_{Client-ServiceProvider}$ session key and makes sure that the message contains the correct value for the timestamp. If that is the case, the client can start interacting with the Service Provider. When the "Service Providers" are the campus-wide printers at a place like Purdue, as in the motivatiional scenario painted at the beginning of this section, it is the $K_{Client-ServiceProvider}$ key that allows a student's laptop to send his/her job directly to a printer over an encrypted connection.

- An additional advantage of separating AS from TGS (although

they may reside in the same machine) is that the Client needs to contact AS only once for a Client-to-TGS ticket and the Client-to-TGS session key. These can subsequently be used for multiple requests to the different service providers in a network.

- In your use of network-based client-server applications, you are likely to run into the acronym GSS-API (sometimes abbreviated GSSAPI) when a server asks you to authenticate yourself. GSS-API is an official standard and Kerberos is the most common implementation of this API. The acronym GSS-API stands for Generic Security Services API. [I suppose you already know that API stands for *Application Programming Interface*. API has got to be one of the most commonly used acronyms in modern engineering.]

10.4: RANDOM NUMBER GENERATION

Secure communications in computer networks would simply be impossible without high quality random and pseudorandom number generation. Here are some of the reasons:

- The session keys that a KDC must generate on the fly are nothing but a sequence of randomly generated bytes. For the purpose of transmission over character-oriented channels (as is the case with all internet communications), each byte in such a sequence could be represented by its two hex digits. So a 128-bit session key would simply be a string of 32 hex digits.

- The **nonces** that are exchanged during handshaking between a host and a KDC (see Section 10.2) and amongst hosts are also random numbers.

- As we will see in Lecture 12, random numbers are also needed for the RSA public-key encryption algorithm. Fundamentally, what RSA needs are prime numbers. However, since there do not exist methods that can generate prime numbers directly, we resort to generating random numbers and testing them for primality.

- As you will see in Lecture 24, you also need random numbers to serve as salts in password hashing schemes. As you will learn in that lecture, you combine randomly generated bits with the string of characters entered by user as his/her password, and then hash the whole thing to create a password hash. Salts make it much more challenging to crack passwords by table lookup.

- You need true random numbers, as opposed to pseudorandom numbers, to serve as one-time keys.

10.4.1: When Are Random Numbers Truly Random?

- To be considered truly random, a sequence of numbers must exhibit the following two properties:

  Uniform Distribution: This means that all the numbers in a designated range must occur equally often.

  Independence: This means that if we know some or all the number up to a certain point in a random sequence, we should not be able to predict the next one (or any of the future ones).

- Truly random numbers can only be generated by physical phenomena (microscopic phenomena such as thermal noise, and macroscopic phenomena such as cards, dice, and the roulette wheel).

- Modern computers try to approximate truly random numbers through a variety of approaches that we will address in Section 10.6 through 10.8 of this lecture.

- Algorithmically generated random numbers are called **pseudo-random numbers**. [Despite the pejorative sense conveyed by "pseudo," the repeatability of pseudorandom numbers is of great importance in engineering work. Let's say you are debugging

a computer program that requires random input for one of its variables. If you only had access to truly random numbers for testing the program, it would be difficult for you to be certain that the change in the behavior of the program for its different runs was not because of a bug in your code.]

10.5: PSEUDORANDOM NUMBER GENERATORS (PRNGs): LINEAR CONGRUENTIAL GENERATORS

- This is by far the most common approach for generating pseudo-random numbers for non-security applications.

- Starting from a seed X_0, a sequence of (presumably pseudorandom) numbers X_0, X_1, ..., X_i, ... is generated using the recursion:

$$X_{n+1} \;=\; (a \cdot X_n \;+\; c) \;\bmod\; m$$

where

m	*the modulus*	$m > 0$
a	*the multiplier*	$0 < a < m$
c	*the increment*	$0 \leq c < m$
X_0	*the seed*	$0 < X_0 < m$

- The values for the numbers generated will be in the range $0 \leq X_n < m$.

- As to how random the produced sequence of numbers is depends critically on the values chosen for m, a, and c. For example, choosing $a = c = 1$ results in a very predictable sequence.

- Should a previously generated number be produced again, what comes after the number will be a repeat of what was seen before. That is because for a given choice of m, c, a, the next number depends only on the current number. Consider the case when $a = 7$, $c = 0$, $m = 32$ and when the seed $X_0 = 1$. The sequence of numbers produced is $\{7,\ 17,\ 23,\ 1,\ 7,\ 17,\ 23,\ 1, ...\}$. The period in this case is only 4.

- Since the "randomness" property of the generated sequence of numbers depends so critically on m, a, c, people have come up with criteria on how to select values for these parameters:

  – To the maximum extent possible, the selected parameters should yield a **full-period sequence** of numbers. The period of a full-period sequence is equal to the size of the modulus. Obviously, in a full-period sequence, each number between 0 and $m - 1$ will appear only once in a sequence of m numbers.

  – It has been shown that when m is a prime and c is zero, then for certain value of a, the recursion formula shown above is guaranteed to produce a sequence of period $m - 1$. Such a

sequence will have the number 0 missing. But every number n, $0 < n < m$, will make exactly one appearance in such a sequence.

– The sequence produced must pass a suite of statistical tests meant to evaluate its randomness. These tests measure how uniform the distribution of the sequence of numbers is and how statistically independent the numbers are.

- Commonly, the modulus m — we want it to be a prime — is chosen so that it is also the largest positive integer value for a system. So for a 4-byte signed integer representation, m would commonly be set to $2^{31} - 1$. With $c = 0$, our recursion for generating a pseudorandom sequence then becomes

$$X_{n+1} = (a \cdot X_n) \bmod (2^{31} - 1)$$

- Earlier we said that when m is a prime and c is zero, then certain values of a will guarantee an output sequence with a period of $m - 1$. A commonly used value for a is $7^5 = 16807$.

- Statistical properties of the pseudorandom numbers generated by using $m = 2^{31}-1$ and $a = 7^5$ have been analyzed extensively. It is believed that such sequences are statistically indistinguishable from true random sequences consisting of positive integers greater than 0 and less than m.

- But are such sequences cryptographically secure? [The previous bullet says that a random sequence produced by a linear congruential generator can be indistinguishable from a true random sequence. So why this question about its cryptographic security? Yes, indeed, taken purely as a sequence of numbers, without any knowledge of how the sequence was produced, the output of a linear congruential generator can indeed look very random when analyzed with probability-based and other statistical tools. But should the attacker know that the sequence was produced by a linear congruential generator, all bets are off regarding its cryptographic security. Read on.]

- A pseudorandom sequence of numbers is **cryptographically secure** if it is difficult for an attacker to predict the next number from the numbers already in his/her possession.

- When **linear congruential generators** are used for producing random numbers, **the attacker only needs three pieces of information to predict the next number from the current number**: m, a, c. The attacker may be able to infer the values for these parameters by solving the simultaneous equations:

$$\begin{aligned} X_1 &= (a \cdot X_0 + c) \bmod m \\ X_2 &= (a \cdot X_1 + c) \bmod m \\ X_3 &= (a \cdot X_2 + c) \bmod m \end{aligned}$$

Just as an exercise assume that $m = 16$, $c = 0$, and $a = 3$.

Assuming X_0 to be 3, set up the above three equations for the next three values of the sequence. These values are 9, 11, and 1. You will see that it is not that difficult to infer the value for the parameters of the recursion.

- The upshot is that even when a pseudorandom number generator (PRNG) produces a "good" random sequence, it may not be secure enough for cryptographic applications.

- A pseudorandom sequence produced by a PRNG can be made more secure from a cryptographic standpoint by restarting the sequence with a different seed after every N numbers. One way to do this would be to take the current clock time modulo m as a new seed after every so many numbers of the sequence have been produced.

10.6: CRYPTOGRAPHICALLY SECURE PRNG'S: The ANSI X9.17/X9.31 ALGORITHM

- As mentioned in the previous section, a pseudorandom sequence of numbers is cryptographically secure if it is difficult for an attacker to predict the next number from the numbers already in his/her possession. The algorithm of the previous section does NOT yield cryptographically secure random numbers.

- We will now talk about a widely used cryptographically secure pseudorandom number generator (CSPRNG). This technique for generating pseudorandom numbers is used in many secure systems, including those for financial transactions, email exchange (as made possible by, say, the PGP protocol that we will take up in Lecture 20), etc.

- X9.17 in the title of this section refers to the 1985 version of the ANSI standard whose Appendix C describes this PRNG. And X9.31 refers to the 1998 version of the standard whose Appendix A2.4 describes the same PRNG.

- As shown in Figure 4, this PRNG is driven by two encryption keys and two special inputs that change for each output number in a sequence.

- Each of the three "EDE" boxes shown in Figure 4 stands for the two-key 3DES algorithm. As you will recall from Lecture 9, the two-key 3DES algorithm carries out a DES encryption, followed by a DES decryption, and followed by a DES encryption. The acronym EDE means "encrypt-decrypt-encrypt".

- The two inputs are: (1) A 64-bit representation of the current date and time (DT_j); and (2) A 64-bit number generated when the previous random number was output (V_j). The PRNG is initialized with a seed value for V_0 for the very first random number that is output.

- All three EDE boxes shown in Figure 4 use the same two 56-bit encryption keys K_1 and K_2. These two encryption keys stay the same for the entire pseudorandom sequence.

- The output of the PRNG consists of the sequence of pairs $(R_j,\ V_{j+1})$, $j = 0,\ 1,\ 2, ...$, where R_j is the j^{th} random number produced by the algorithm and V_{j+1} the input for the $(j+1)^{th}$ iteration of the algorithm. From Figure 4 the output pair $(R_j,\ V_{j+1})$ is given by

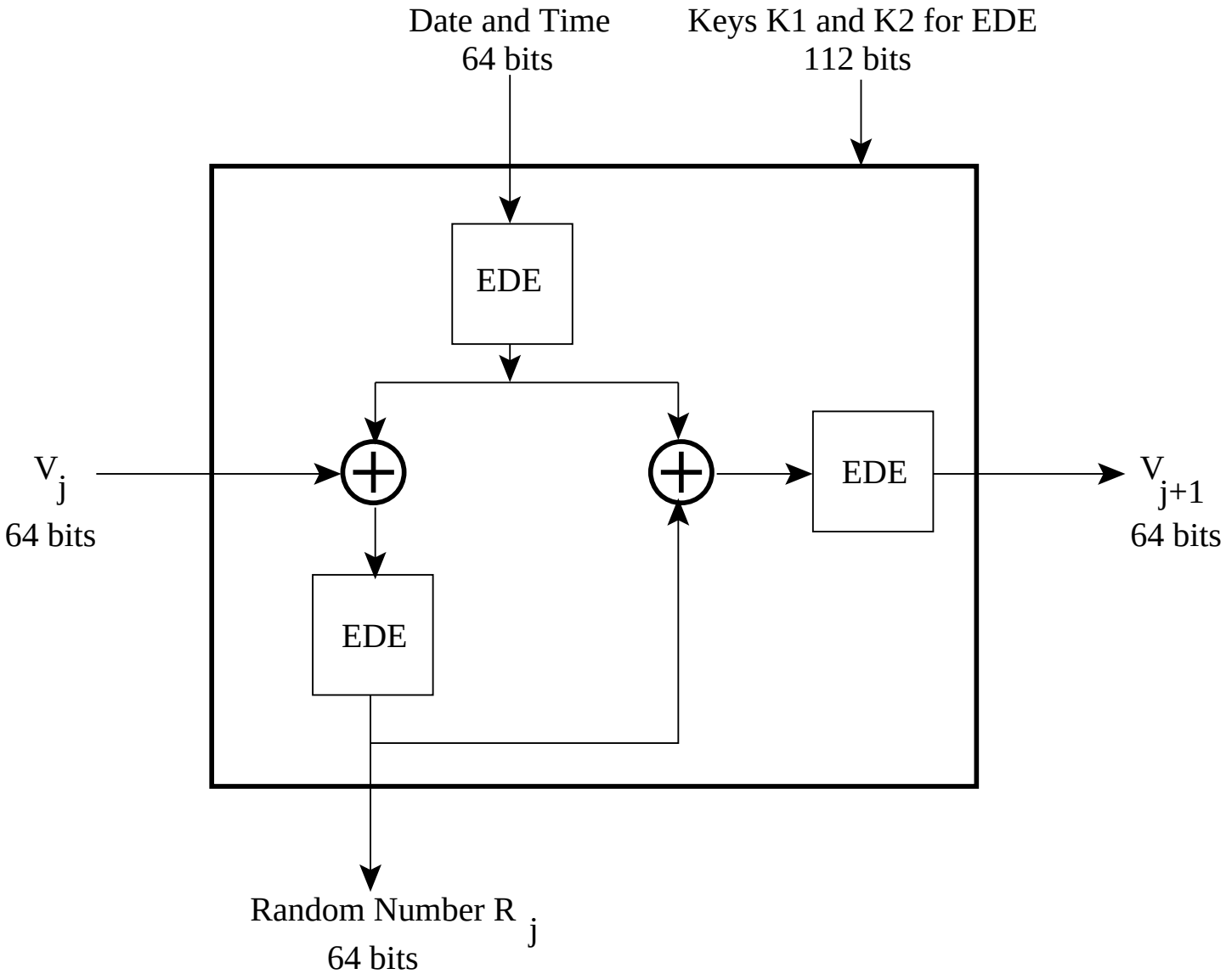

Figure 4: *ANSI X9.17/X9.31 Pseudorandom Number Generator.* (*This figure is from Lecture 10 of "Computer and Network Security" by Avi Kak.*)

$$R_j = EDE\left([K_1, K_2]\,,\ [V_j \otimes EDE([K_1, K_2]\,,\ DT_j)]\right) \quad (1)$$
$$V_{j+1} = EDE\left([K_1, K_2]\,,\ [R_j \otimes EDE([K_1, K_2]\,,\ DT_j)]\right) \quad (2)$$

where $EDE([K_1, K_2],\ X)$ refers to the encrypt-decrypt-encrypt sequence of 3DES using the two keys K_1 and K_2.

- The following reasons contribute to the cryptographic security of this approach to PRNG:
  - We can think of V_{j+1} as a **new seed** for the next random number to be generated. This seed cannot be predicted from the current random number R_j.
  - Besides the difficult-to-predict pseudorandom seed for each random number, the scheme uses one more independently specified pseudorandom input — an encryption of the current date and time.
  - Each random number is related to the previous random number through multiple stages of DES encryption. An examination of Equation (1) above shows there are **more than two EDE encryptions** between two consecutive random numbers. If you could say from Equation (2) that there exists one EDE encryption between a random number and the seed for the next random number, then it would be fair to say that

there exist **three EDE encryptions** between two consecutive random numbers. Since one EDE encryption amounts to three DES encryptions, we can say that there exist **nine DES encryptions** between two consecutive random numbers, making it virtually impossible to predict the next random number from the current random number.

– Even if the attacker were to somehow get hold of the current V_j, it would still be practically impossible to predict V_{j+1} because there stand at least two EDE encryptions between the two.

- Is there a price to pay for the cryptographic security of ANSI X9.17/X9.31? Yes, it is a much slower way to generate pseudo-random numbers. That makes this approach unsuitable for many applications that require randomized inputs.

- Finally, note that whereas the ANSI X9.17/X9.31 standard requires the 2-key 3DES (that is, EDE) in Figure 4, the NIST version allows AES to be used for the same.

- A wonderful article to read on the cryptographic security of PRNGs is "Cryptanalytic Attacks on Pseudorandom Number Generators" by John Kelsey, Bruce Schneier, David Wagner, and Chris Hall.

10.7: CRYPTOGRAPHICALLY SECURE PRNG'S: THE BLUM BLUM SHUB GENERATOR (BBS)

- This is another cryptographically secure PRNG. This has probably the strongest theoretically proven cryptographic security.

- The BBS algorithm consists of first choosing two large prime numbers p and q that both yield a remainder of 3 when divided by 4. That is

$$p \equiv q \equiv 3 \pmod 4$$

  For example, the prime numbers 7 and 11 satisfy this requirement.

- Let

$$n = p \cdot q$$

- Now choose a number s that is relatively prime to n. (This implies that p and q are **not** factors of s.)

- The BBS generator produces a pseudorandom sequence of bits B_j according to

$$
\begin{aligned}
X_0 &= s^2 \mod n \\
\\
for\ i &= 1\ to\ \inf \\
X_i &= (X_{i-1})^2 \mod n \\
B_i &= X_i \mod 2
\end{aligned}
$$

- Note that B_i is the least significant bit of X_i at each iteration.

- Because BBS generates a pseudorandom bit stream directly, it is also referred to as a **cryptographically secure pseudorandom bit generator** (CSPRBG).

- By definition, a CSPRBG must pass the **next-bit test**, that is there must not exist a *polynomial-time algorithm* that can predict the k^{th} bit given the first $k-1$ bits with a probability significantly greater than 0.5. BBS passes this test. [In the theory and practice of algorithms, *polynomial-time* algorithms are considered to be efficient algorithms and *exponential-time* algorithms considered to be inefficient. At Purdue, our class ECE664 goes into such distinctions between the different types of algorithms.]

- The above discussion is not meant to imply that you can only generate pseudorandom single-bit streams with the BBS algorithm. By packing the single bits into, say, 4-byte memory blocks, one

can generate 32-bit integers that would be cryptographically secure. In fact, this is what you are supposed to do in one of the programming problems in the Homework section of this lecture.

10.8: ENTROPY SOURCES FOR GENERATING TRUE RANDOM NUMBERS

- Over the years, new types of random number generators have been developed that allow for the generation of true random numbers, as opposed to just pseudorandom numbers. We will refer to an entity that allows for the production of true random numbers as TRNG for True Random Number Generator. And, as you know, the acronym PRNG stands for a Pseudo Random Number Generator. And the acronym CSPRNG stands for a cryptographically secure PRNG.

- A fundamental difference between a PRNG and TRNG is that whereas the former must have a seed for initialization, the latter works without seeds. This fundamental difference between a PRNG and TRNG also applies to the difference between a CSPRNG and TRNG.

- These new types of random number generators are based on the idea that only the analog phenomena can be trusted to produce truly random numbers. We are talking about analog phenomena

such as thermal noise in electronic components; direct and indirect consequences of human interactions with the computers and computer networks; various system properties that change with time in unpredictable ways; etc. [To be sure, we have always had true sources of random bits that depended on greatly amplifying the thermal noise in resistors and then digitizing it for the production of random bits. But those random bit generators consumed much power and were generally not considered appropriate for routine communication devices in computer networks. The new types of hardware implementations that I mention in this section do NOT suffer from this limitation.]

- We will consider an *entropy source* to be any source that is capable of yielding a truly random stream of 1's and 0's. Presumably, the randomness of the bits provided by the entropy sources is, directly or indirectly, a consequence of some analog phenomenon.

- The reader may ask: If we can have entropy sources for the production of random sequences of 1's and 0's, why bother with CSPRNGs of the type I presented in Section 10.6?

- To answer the above question, entropy sources, in general, are not capable of providing random bits at the rate needed by high-performance applications. For such applications, the best they can do is to serve as the seeds needed by CSPRNGs of the type presented in Section 10.6.

- The use of entropy-source based random numbers for security in computer networks has spawned new phrases that are now part

of the lexicon of network security:

- *"entropy source"*
- *"hardware entropy source"*
- *"software entropy source"*
- *"accumulation of entropy"*
- *"eating up entropy"*
- *"entropic content"*
- *"extent of entropy"*
- *"entropy hole"*
- etc.

Of course, "entropy" itself is a very old idea and, in the information theoretic context, measures the extent of uncertainty one can associate with a random process. Nevertheless, before the advent of the new class of random number generators described in this section, you were unlikely to run into phrases like "*the keys generated by a communication device may be weak because they are based on insufficient accumulation of entropy.*"

- If we organize the bit stream produced by an entropy source into words, which could be bytes, and if we consider each such word as a random variable that can take the i^{th} value with probability p_i, we can associate the following entropy with the bit stream:

$$H \quad = \quad -\sum_i p_i \log_2 p_i$$

Let's say the bit stream is organized into bytes. A byte takes on 256 numeric values, 0 through 255. If each of these values

is equally probable, then $p_i = \frac{1}{256}$, and the entropy associated with the entropy source would be 8 bits. This is the highest entropy possible for 8-bit words. If the probability distribution of the values taken by 8-bit words were to become nonuniform, the entropy will become less than its maximum value. For the deterministic case, when all the 8-bit patterns are the same, the entropy is zero.

- Let's say you want a random number that can be used as a 128-bit key. Ideally, you would want your entropy source to produce 128-bit words with equal probability. Such an entropy source has an entropy of 128 bits.

- Before delving into the nature of the modern entropy sources, my immediate goal is to re-emphasize the importance of randomness to the security of modern computer networks. As you will realize from the brief discussion in the next bullet (and as you'll realize even more strongly later in this course), if a network device were to use a poor quality random number generator — one whose random numbers are predictable — it would be much too vulnerable to security exploits. The more nonuniform the probabilities of the values taken by the random numbers, the more predictable they become.

- Ideally, any network device — be it a computer, a router, or, for that matter, an embedded device with a communication inter-

face — would only want to use one-time random numbers for the keys needed for encrypting the communications with other hosts or devices. A one-time random number means that there is very little chance that the same random number will be used again in the foreseeable future. One-time random numbers obviously translate into one-time keys. A network device may need session keys as we mentioned earlier in this lecture or public/private keys along the lines talked about in Lecture 12. Whereas a sequence of random bytes can be used directly as a session key, the public/private keys are obtained from those random bytes that can be shown to constitute prime numbers (see Lecture 12). **If a random number generator is so poor that it can only generate one of a small number of different random numbers, its session keys become predictable. As you will see in Section 12.6, such a random number would also result in an attacker being able to figure out the private key that goes with a public key.** [It is important to bear in mind that even an algorithmic approach to random number generation, of the sort described in Section 10.6, needs an initialization number to get it started. To the extent this initialization number is not truly random for each execution of the algorithm, the random number you get from the algorithm may not be as cryptographically secure as you might think.]

- Let's get back to the subject of entropy sources for the production of random bits and bytes. There are two types of entropy sources to consider: the on-chip hardware based entropy sources and the other purely software based entropy sources.

- The on-chip hardware based TRNG obviously use hardware entropy sources, as you'll see in what follows in this section. On the other hand, a software based TRNG uses different types of software processes as sources of entropy, as you will see in Section 10.9.

- The on-chip hardware based approach is exemplified by Intel's Bull Mountain Digital Random Number Generator (DRNG). It uses two inverters (an inverter converts an input of 0 into an output of 1 and vice versa) with the output of one connected to the input of the other. This manner of connecting the two inverters means that, unless the conditions external to the inverters force their outputs to be otherwise, the output of one inverter must be opposite of the output of the other. These external conditions are controlled by a driver circuit. In the off state of the driver circuit, when the output of one inverter is 1, the output of the other must be 0. As to which inverter would output a 1 and which would output a 0 depends on the thermal noise that accompanies the 1-to-0 and 0-to-1 transitions of the circuit elements. In theory, the two inverters must be exactly identical for the stream of 1's and 0's produced in this manner to be truly random. Since that is impossible to satisfy in practice, additional circuity must be used to compensate for any departure from the ideal in the two inverters. Intel has shown that this approach can produce a bit stream at 3 GHz. This bit stream must subsequently be conditioned to compensate for any biases in randomness caused by the two inverters not being truly identical. Finally, the con-

ditioned bits are used to initialize a hardware implementation of a CSPRNG for higher production rates of the random bytes. Intel also provides a machine-code instruction, `RDRAND`, for 64-bit processors for fetching random numbers from the DRNG. [At this point it is important to mention that even the best entropy sources are performance constrained with regard to how fast they can generate the random numbers. By its very definition, an entropy source must sample some analog phenomenon. So the rate at which an entropy source can produce the random bits depends on the rate at which the analog phenomenon is changing. Even though Intel's hardware based approach generates truly random bits faster than any of the other approach I'll mention later, it must nonetheless be used with a hardware implemented CSPRNG for producing bytes at the rates needed by various applications.]

- The next section takes up the subject of software entropy sources for the production of truly random bits.

10.9: SOFTWARE ENTROPY SOURCES

- The previous section introduced the notion of entropy sources for generating true random numbers and focused specifically on hardware sources of entropy. In this section, we take up the subject of software entropy sources.

- Software entropy sources are based on the fact that in virtually every computer there are constantly occurring "phenomena" that, either directly or indirectly, are consequences of some human interaction with that computer or some other networked computer. For example, the exact time instants associated with your keystrokes as you are working on your computer is a random process with a great deal of uncertainty associated with it. [If you are like the rest of the human beings, after every few keystrokes you are either looking at what you just entered to make sure that you did not make any errors, fetching yourself a cup of coffee, watching the newspaper that's open in another window, pacing the floor back and forth if you are stuck in the middle of a difficult writing assignment, and so on. All of these are analog sources of randomness that translate into randomness associated with your keystrokes.] By the same token, the timing of the interrupts generated by you clicking on your mouse buttons are also random. Equally random are the movements of the mouse pointer on your screen. Yet another source of entropy are the times associated with the disk I/O events.

- Other software sources of entropy include information entered in various log files (in `/var/log/syslog`, for example, that is used for the logging of networking and security events), and the output of various system commands such as `ps`, `pstat`, `netstat`, `vmstat`, `df`, `uptime`, etc.

- All of these software sources of entropy can be divided into two categories: those that can only be accessed with root privileges (these are referred to as belonging to the *kernel space*) and those that are accessible with ordinary user privileges (these are referred to as belonging to *user space*).

- The random bits made available by the kernel space entropy sources are available through a special file `/dev/random` in your Linux/Unix platforms.

- On the other hand, the random bits made available by user space entropy sources can be obtained either through EGD (Entropy Gathering Daemon) or through PRNGD (Pseudo Random Number Generator Daemon).

- In the three subsections that follow, I'll first take up `/dev/random`, and its closely related `/dev/urandom` as sources of random bits. Subsequently, I'll talk about EGD and PRNGD as user-space entropy suppliers.

10.9.1: /dev/random and /dev/urandom as Sources of Random Bytes

- As mentioned previously, `/dev/random` gathers entropy in the kernel space. It is based on the randomness associated with keystrokes, mouse movements, disk I/O, device driver I/O, etc.

- You might wonder how much entropy such a source can produce per unit time. What if you are not banging on the keyboard, or playing with mouse, or fetching anything from the disk through a job running in the background (if your job was running in the foreground, then you'd be banging on the keyboard, won't you!), etc., would there still be sufficient entropy generated by `/dev/random` for a 256-bit key that your network interface needs to send some system-generated message to remote machine securely?

- To respond to the question posed above, yes, it is possible for `/dev/random` to block until its pool of random bits possesses sufficient number of bits at the entropy level you want.

- Software sources of entropy can typically only generate a few hundreds bits of entropy per second. So if your needs for random bytes exceeds this rate, you obviously cannot rely on `/dev/random`.

- For a non-blocking kernel space source of entropy, you can use `/dev/urandom` that uses the random bits supplied by `/dev/random` to initialize a CSPRNG (see Section 10.6) in order to produce a very high-quality stream of pseudorandom bytes. Being pseudo-random, the byte stream produced by `/dev/urandom` will obviously have less entropy than the byte stream coming from `/dev/random`.

- In order to use `/dev/random`, it is sometimes important to also examine the directory `/proc/sys/kernel/random/`. This directory contains text files with information on the entropic state of what you can expect to see if read the special file `/dev/random`. For example, the file `entropy_avail` contains an integer that is the value of the entropy of the sequence of 1's and 0's in the entropy pool. The size of the entropy pool can be read from the file `poolsize` in the same directory.

- Shown below is a Perl script whose inner `for` loop queries the file `entropy_avail` once every second until the entropy exceeds the threshold of 32. Subsequently, the script calls `sysread()` and attempts to read 16 bytes from `/dev/random`. However, as you will see in the output that follows the script, the actual number of bytes harvested from `/dev/random` depends on the entropy of what is in the entropy pool at the moment.

```
#!/usr/bin/perl -w
```

```
##  UsingDevRandom.pl
##  Avi Kak
##  April 22, 2013

use strict;

open FROM, "/dev/random" or die "unable to open file: $!";
binmode FROM;
for (;;) {
    my $entropy = 0;
    for (;;) {
        $entropy = `cat /proc/sys/kernel/random/entropy_avail`;
#       last if $entropy > 128;
        last if $entropy > 32;
        sleep 1;
    }
    my $pool_size = `cat /proc/sys/kernel/random/poolsize`;
    my $how_many_bytes_read = sysread(FROM, my $bytes, 16);
    print "Number of bytes read: $how_many_bytes_read\n";
    my @bytes  = unpack 'C*', $bytes;
    my $hex = join ' ', map sprintf("%x", $_), @bytes;
    my $output = sprintf "Entropy Available: %-4d    Pool Size: %-4d      \
Random Bytes in Hex: $hex",
                  $entropy, $pool_size;
    print "$output\n\n\n";
    sleep 1;
}
```

- Shown below is a small segment of the output produced by the outer infinite loop in the script:

```
Number of bytes read: 16
Entropy Available: 128     Pool Size: 4096      Random Bytes in Hex: ed a9 6f 82 d9 74 c8 3 10 9c 44 d3 bc cb 4

Number of bytes read: 14
Entropy Available: 113     Pool Size: 4096      Random Bytes in Hex: a2 2a ad cb 8f 84 80 12 db 6a 2e 50 fc e2

Number of bytes read: 13
Entropy Available: 105     Pool Size: 4096      Random Bytes in Hex: 38 2a f8 14 a 5b 47 1a ec b4 b1 2a b9
```

```
Number of bytes read: 8
Entropy Available: 42      Pool Size: 4096      Random Bytes in Hex: 5e 25 9e 80 69 b5 4d 34

Number of bytes read: 13
Entropy Available: 110     Pool Size: 4096      Random Bytes in Hex: 9a e0 4b a4 7e 8e e6 c8 67 9c d5 7a 7

Number of bytes read: 10
Entropy Available: 86      Pool Size: 4096      Random Bytes in Hex: 36 9d ea ac 22 4e 9 d9 7c a1

Number of bytes read: 8
Entropy Available: 62      Pool Size: 4096      Random Bytes in Hex: 66 9b b d8 2f 31 af 99

Number of bytes read: 8
Entropy Available: 41      Pool Size: 4096      Random Bytes in Hex: 88 28 64 a5 a2 41 38 3a

Number of bytes read: 13
Entropy Available: 108     Pool Size: 4096      Random Bytes in Hex: 6a 77 56 1 29 eb 1d 2c 84 ee 43 18 49

Number of bytes read: 15
Entropy Available: 121     Pool Size: 4096      Random Bytes in Hex: 6 fc 97 67 28 a1 9 d9 2f e8 63 a3 36 2c 56
```

- An important thing to note about this output is that every once in a while it appears to hang. However, just by moving your mouse a bit or entering a few keystrokes gets the output going again. That is further proof of the fact that this kernel-space entropy source gets its randomness from the keystrokes and the mouse movements.

- The act of reading `/dev/random` depletes the random bit pool of the bytes that are read out and causes a reduction in the entropy of what is left behind. The '`sleep 1`' statement at the end of the script is to allow for the replenishment of the entropy before we examine the pool again.

- `/dev/random` and `/dev/urandom` were created by Theodore Ts'o.

10.9.2: EGD — Entropy Gathering Daemon

- As mentioned previously in Section 10.9, EGD gathers its entropy from user-space events. So if for some reason you do not have `/dev/random` in your machine, you could try to install EGD that is available from SourceForge.

- If you download the source code for EGD from SourceForge and examine its implementation code (it is in Perl), you will see the following system commands (amongst several others) that yield the textual output that serves as the starting point for collecting entropy:

```
vmstat -s                 # print virtual memory statistics

netstat -in               # print network connections, routing tables, etc.

df                        # display disk space available

lsof                      # list open files

ps aux                    # snapshot of the current processes

ipcs -a                   # provide info on interprocess communications

last -n 50                # show listing of the last 50 logged in users

arp -a                    # show MAC address of network neighbor
```

- Associated with each source of entropy as listed above is a filter, referred to as `filter` in the EGD source code, that when set

to 1 implies that we ignore all non-numerical output from the command. In other words, all of the output of a command is accepted only for the cases when `filter` is set to 0.

- EGD associates with each source a parameter denoted `bpb`, which stands for "bits of entropy per byte of the output". For example, the `bpb` parameter associated with the source '`vmstat -s`' is 0.5. What that means is that each byte of source (after we remove all non-numerical characters since the value of `filter` for this source is 1) is known to yield an entropy addition of only 0.5 bits. Presumably that implies that if we can extract two bytes of just numerical information from this source and add that to the entropy pool, we can increase the entropy of the pool contents by 1 bit.

- As I mentioned earlier, the listing of the sources I show above is a subset of all the sources in EGD. If any of the sources is found to be "dead", in the sense of not yielding any returns, it is dropped from the source list.

- You fire up the server daemon by calling

```
egd.pl ~/.gnupg/entropy
```

where `egd.pl` is the main Perl file for EGD in the installation directory. The argument to the command creates a Unix domain server socket named `entropy` in the `.gnupg` file of your home directory. [There is nothing sacrosanct about either the name of the Unix domain socket or its location.

Additionally, you are also allowed to use a TCP server sockets. If you choose to use a TCP server socket, the argument to `egd.pl` would be something like `localhost:7777` assuming you want the server to monitor the port 7777.] Subsequently, the entropy daemon will start serving out the random bytes through the Unix socket named by the argument.

- Since EGD invoked in the manner shown above serves out its random bytes through a server socket, you need to create a client socket to receive the random bytes from the server. In order to get used to EGD, the easiest thing to do at the beginning is to use the `egc.pl` client in the `example` subdirectory in the installation directory. Here is a listing of some of the commands you could invoke in the examples directory:

```
egc.pl ~/.gnupg/entropy get             # returns the bits currently in the pool

egc.pl ~/.gnupg/entropy read 16         # fetches and displays 16 random bytes

egc.pl ~/.gnupg/entropy readb 16        # fetches 16 random bytes (blocking)
```

Note the difference between the last two invocations of the client `egc.pl`. The third invocation blocks if you ask for more bytes than there is entropy in the pool. It blocks until the entropy increases to the level commensurate with the number of bytes you requested. The second invocation, on the other hand, is nonblocking because it uses the currently available random bytes to seed a CSPRNG to yield guaranteed number of bytes (whose entropy would obviously be lower than what you would get for the same number of bytes returned by the third invocation).

- Typically, you can count on this entropy server to generate roughly 50 bits of entropy per second.

- You can stop the server daemon by the command '`killall egd.pl`'

- EGD was created by Brian Warner.

10.9.3: PRNGD (Pseudo Random Number Generator Daemon)

- With regard to the basic mechanism used for gathering entropy, PRNGD is very similar to EGD, in the sense that the former also uses the output of user-space processes for randomness.

- While the basic mechanism for entropy gathering is the same, PRNGD uses its own set of user-space commands for the random output it needs to generate entropy. In addition to some of the same system commands as used be EGD, PRNGD also uses the output obtained by invoking the `stat` (for status) command on system files that are likely to be accessed frequently. Examples of such files are `/etc/passwd`, `/tmp`, etc. PRNGD also makes calls to `times()`, `gettimeofday()`, `getpid()`, etc., for additional random outputs.

- One large differences between PRNGD and EGD is that the former is in C whereas the latter is in Perl.

- The other large difference lies in the fact that PRNGD uses the random bits collected from its entropy sources to seed a CSPRNG — more specifically the OpenSSL PRNG — and you only see the output of the CSPRNG. So, at least theoretically speaking, you

never see truly random bytes with PRNGD. On the plus side, though, you will not run into blocking reads of the bytes with PRNGD.

- PRNGD was created by Lutz Janicke.

10.9.4: A Word of Caution Regarding Software Sources of Entropy

- First of all, you need to know that the use of software sources of entropy is more common than you think.

- As alluded to earlier in Section 10.9, random numbers are needed not just by your computer when you log into a remote server using the SSH protocol or when your computer is trying to authenticate the server at an e-commerce site like Amazon. Random numbers are also needed by what are known as *headless devices*, these being routers, firewalls, sever management cards, etc., for establishing secure communications with other hosts in a network.

- As it turns out, a very large number of such headless devices use software entropy sources for the random bytes they need for the keys. The most common such source is `/dev/urandom` that is guaranteed to provide you with any number of pseudorandom bytes in a non-blocking fashion.

- However, as pointed out in Section 10.9.1, the output bytes produced by `/dev/urandom` are NOT meant to be truly random since they are produced by a CSPRNG that, in turn, is seeded by the output of the more truly random `/dev/random`. The headless de-

vices are not able to use `/dev/random` directly because its output blocks until sufficient entropy has built up to deliver the number of bytes needed.

- The main problem with `/dev/urandom` occurs at boot up. For obvious reasons, the very first thing a communication device would want to do would be to create the keys it needs to communicate with other hosts. However, that's exactly the moment when a software entropy source like `/dev/random` is likely to be in an *entropy hole*, that is, likely to possess very little entropy. Therefore, any bytes produced by `/dev/urandom` at this juncture are likely to be low-entropy bytes, which would make them predictable.

- In a recent publication by Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman, it was demonstrated that this weakness in the generation of random numbers in headless devices in the internet allowed them to compute the private keys for 0.5% of the SSL/TLS hosts and 1.06% of the SSH hosts from a sampling of over 10 million hosts in the internet. The publication is titled "Mining your Ps and Qs: Detection of Widespread Weak Keys in Network Devices," that appeared in *Proc. 21st USENIX Security Symposium, 2012*. The "P" and "Q" in the title refer to the two prime factors of the modulus used in the RSA algorithm that I'll present in Lecture 12.

- As to how Heninger et al. managed to figure out the private keys

used by a large number of communication devices in the internet is explained in Section 12.7 of Lecture 12.

10.10: HOMEWORK PROBLEMS

1. What aspect of the Needham-Schroeder Key Distribution Protocol gives each of the two parties A and B (who want to communicate securely with each other) the confidence that no third party C is masquerading as the other?

2. What is a nonce and why is it used in the Needham-Schroeder protocol?

3. What sort of secure communication applications is the Kerberos protocol intended for?

4. What does the acronym GSS-API stand for and what is its relationship to Kerberos?

5. What is the difference between algorithmically generated random numbers and true random numbers?

6. What are the essential elements of the X9.17/X9.31 algorithm

for generating pseudorandom numbers that are cryptographically secure?

7. **Programming Assignment:**

Write a Python or a Perl script that generates a cryptographically secure sequence of 8-bit unsigned integers using the Blum-Blum-Shub algorithm of Section 10.7. The algorithm as described generates a bit stream. You would need to pack the bits into one-byte bit arrays. If using Python, take advantage of the bit shifting functions provided in the BitVector class for packing the pseudo-random bits into 8-bit BitVectors. [Since you do not yet know how to generate prime numbers, you will have to supply to your script the two primes p and q that must both be congruent to 3 modulo 4. At this time, fetch the primes you need from one of the several web sites that publish a large number of prime numbers. Later on, after Lecture 11, you will be able to generate your own primes for the script here.]

8. **Programming Assignment:**

For a stream of 100,000 bytes, compare the execution time of the program you wrote for the previous problem with that for your implementation of the RC4 algorithm for doing the same. If you are using Python, it is rather easy to measure the execution time with the `timeit` module. [Pages 322 and 333 of the "Scripting With Objects" book illustrate how you can use Python's `timeit` module.] If using Perl, you can use either the builtin function `time()` or, better yet, the `Benchmark` module for doing the same. Which algorithm, BBS

or RC4, is faster for generating the byte stream and why?